Paxos made moderately working

May 2020

Contents

1 Introduction

Distributed systems theory is really a troublesome field.
The main idea is that we want a set of multiple computers to behave like there was only one.
For example, we want our database to be split among multiple machines to avoid issues like data losses, service interruption, server overload, storage issues and many other fun problems. Basically we want to improve the performances of a system by getting it to run on multiple machines.
This implies that all the machines composing our system needs to stay synchronized.
There are 3 main components that we want to maintain:

- Consistency: every machines in the system as to agree with each other, you would not want a database that doesn't respond the same thing depending on the actual machine that received the request.
- Availability: I want my system to be always able to respond, I don't want service interruptions
- Partition tolerance: This is a tricky one but basically I want my system to work as intended even if some machines are separated from each other (when a partition occurs). For example if I have 3 machines (A, B and C) in my system, and it turns out that A and B can access each other but can no longer access C (but C is still up, it is just not accessible), then a partition occured and I don't want my system to diverge on the two sides of the partition because they are not able to communicate.

This looks really interesting but one of the first theorem you will learn while working on distributed systems is that it is impossible to maintain these 3 components at once.
Many algorithms have addressed this issue by making concessions. They are also many paper that explains the problems in further details as well as adding other ones (Byzantine generals, FLP ...) but we will not talk about these.
We will talk about one legendary algorithm called Paxos, described in The Part-Time Parliament.
This algorithm is known for being really hard to understand but also because its original research paper tells a story to describe the algorithm.
I really wanted to understand this algorithm, so I reimplemented it in C++.
You can find the code here.
The present article acts as a guide to navigate between the implementation and the original paper.

2 Consistency and progress condition

We will first talk of the key parts of the algorithm and how they are presented in the paper.
The author created a fictionnal civilisation on the island of Paxos to tell his story. The culture of the paxons (residents of Paxos) is really particular because they have the bad habit of leaving whatever they are doing to do their personnal activities.
Even the legislators in the parliament could leave in the middle of a vote. Yet paxons had laws, and those laws had to be written somewhere after being voted. How could the legislators agree with each other on the current state of the law, and how could they create a new law?
These two problems are respectively called the "Consistency" and "Progress" conditions in the paper.
Of course, this is an image for a distributed system. The system is the parliament and each legislator is a machine. The fact that a legislator might leave the parliament for his personnal activities represents the fact that the machine either crashed or is not accessible anymore. Yet, each legislator had to know all the laws in place at the current time and must be able to vote for a new one.
The authors then describes how paxons represented the law. The law is defined by a sequence of numbered decree, for example:

- 132: Lamps must use only olive oil
- 155: The olive tax is 3 drachmas per ton

To record the law, one might want to hire a secretary to register every decree, but in Paxos no one was willing to act as a secretary and stay in parliament for the entire session.
Instead, each legislator kept a ledger containing the decrees that were passed. Each legislator had its own version of the law.

We can now better rephrase the consistency condition stated earlier: No two ledgers can contain contradictory information. It means that if a legislator has the decree 132 in the above example, no other legislator can have a different entry for decree 132.
However, it was possible for a legislator to not have any entry for decree 132, for example if he didn't learned about it yet.
Of course, the consitency condition could be trivially fulfilled by leaving every ledger blank so we needed the progress condition.
In paxos, an atmosphere of mutual trust prevalled, it means that legislators were willing to accept any decree proposed by any other legislator.
This might cause some consistency issues. For example, a group of legislators pass the decree **37: Painting on temple walls is forbidden**. And then leave the parliament before another group of legislators enters and pass (without knowing anythin about the previously passed decree) the decree **37: Freedom of artistic expression is guaranteed**.
To solve this, we need at least one legislator in the first group to also be in the second, so he can warn them that this decree number already exists.
To make sure that any two groups of legislators have at least one legislator in common, we simply need to make sure that both of them are a majority. This means that a decree can only be passed if a majority of legislators are in the parliament. This guarantee allows to define the progress condition as follow: If a majority of the legislators were in the parliament, then any decree proposed by a legislator in the the parliament would be passed and every decree that had been passed would appear in the ledger of every legislator in the parliament.

We now need to maintain those 2 conditions.

3 Assumptions

The article then states a few assumptions for the organisation of the parliament.
First, every legislators has a personnal ledger, in which they write decrees.
This ledger represents any kind of persistent data storage, in our implementaion, it will simply be a file, but it could an actual database.

It is stated that legislator could take notes at the back of their ledgers to not forget what they were doing after leaving the parliament. We will also use files for this matter, we will use them to keep track of data relative to the algorithm that MUST NOT be lost uppon crashes.
Legislators would write other notes on a slip of paper which he might (or not) lose uppon leaving the parliament. This represents simple memory storage.
Legislators had an hourglass to measure time, it means that our servers must have internal clocks, in our implementation we will not use them as the complete protocol requires because it isn't usefull in the case where we are initiating decrees manually (which we do in our implementation).

It is then specified that the parliament had bad acoustics, the only way for legisltors to communicate was to send messengers. Messengers could leave the parliament for personnal activities the same way legislators did. It means that our machines cannot have direct inter-process communication, we will here use TCP connections. The application layer of our message will be further detailed when will talk of the protocol.

4 Design concerns

Let's put the basis of our implementation, we will only detail here what is relevant for the protocol, all the socket, events or configuration isn't interesting here so we will only focus on the protocol in itself by making abstraction of everything else.

Let's build our first class, the Legislator, as stated in the assumptions it contains a ledger and some volatile data for the algorithm but we do not know them yet.

```
class Legislator
{
public:
    Legislator(const LegislatorConfig& config);
    LegislatorConfig config_;

private:
    Ledger ledger;
};
using shared_legislator = std::shared_ptr<Legislator>;

extern shared_legislator self;
extern std::unordered_map<std::string, shared_legislator> legislators;
```

The LegislatorConfig simply hold the ip and port of the server, as well as a name to identify it.
Legislator are created with the factory design pattern using a configuration file, it simply contains configurations of all the legislators in the parliament.
The name of the legislator currently running is given via the command line.
So if you want to simulate a parliament with 2 legislators, you need to the program twice with the same configuration file, but by putting a different name (that must be both present in the configuration file).
There is nothing really worth noting in the Ledger class, it simply contains a bunch of methods to retrieve data (for example decrees).
You will also notice 2 global variables, a map between the name of the legislator and the legislator itself, and a pointer to the legislator that corresponds to that particular machine.

We will add a lot of methods in this class, it is really our central point.

We then have the Decree class.
It is simply a value object containing an int. The decree will only be an int here.

5 The single-decree synod

5.1 Mathematical results

We now have everything we need to start the algorithm, the paper presents it in 2 steps.
First there is the single-decree synod protocol and then the multi-decree parliament protocol which directly derives from the first. The single-decree protocol is itself a derivation of multiple sub-protocols. There is first the preliminary protocol, that is the mathematical basis of the algorithm, then the basic protocol that derives directly from it that respects the consistency condition and then the complete protocol that adds the progress condition.

The goal of the single-decree protocol is to synchronize a single decree between all our legislators. It is worth to note that the proposed implemention is actually only the basic single-decree protocol, the multi-decree part is yet to be done and the complete single-decree protocol is irrelevant in our implementation.

The decree is chosen through a series of numbered ballots.
A ballot is simply a referendum on a single decree, in each ballot each legislator has the choice to either vote or to not vote for the decree.
To each ballot was associate a non-empty set of legislators called the quorum, the ballot succeeded (and the decree was passed) if and only if every legislator in the quorum voted for it. A ballot B is said to be earlier than a ballot C if B's ballot number is inferior to C's.

Let's now consider a set β of ballots, we will see that by adding a few conditions, once a ballot has succeeded, every later ballots will be fore the same decree, thus satisfying the consistency condition.

The consistency condition is satisfied if:

- $B1(\beta)$: Each ballot in β has a unique ballot number
- $B2(\beta)$: The quorum of any two ballots in β have at least one legislator in common
- $B3(\beta)$: For every ballot B in β, if any legislator in B's quorum voted in an earlier ballot in β, then the decree of B equals the decree of the latest of those earlier ballots

Rather than going into the maths of why this works I will try to build a strong intuition of this.

So let's consider ballot B with number 4 for decree D. Given the success of B, every legislators in B's quorum voted for it by definition of a successful ballot.
Let's now consider ballot C with number 5 whose decree is yet to be determined. C's quorum contains at least one legislator in common with B's quorum (because of $B2(\beta)$).
Since every legislator in B's quorum voted in B for decree D, we can immediatly deduct that at least one legislator in C's quorum previously voted in B for decree D. B being the earliest ballot before C, the decree of C MUST be D to satisfie $B3(\beta)$.

Take a moment to think about it, it means that once a ballot succeeded, every later ballot will be for the same decree, if a legislator was not in the parliament at the time the decree was originally passed, he would still learn its existence thanks to the new ballots, holding the same decree.

5.2 The preliminary protocol

We can derive a protocol from the requirement that $B1(\beta) - B3(\beta)$ remain true where β is the set of all ballots that were or are being conducted.
Each legislator could initiate a ballot by choosing its number, decree and quorum. The choice derive directly from the need to maintain $B1(\beta) - B3(\beta)$.

To maintain $B1(\beta)$, each ballot needs to receive a unique number. A legislator can take notes in his ledger to remember of which ballot he already initiated to avoid picking the same number twice. To avoid two different legislators from initiating ballots with the same number, the set of possible ballot number is partitionned among the legislators. There are multiple way of doing this, in our implementation we give ballot numbers to legislators following a round-robin basis.
If we have 3 legislators (A, B and C), legislator A will have ballot numbers 0, 3, 6 ...
legislator B will have 1, 4, 7 ... and legislator C will have 2, 5, 8 ...

To maintain $B2(\beta)$ we can simply choose the quorum as any majority set among the legislators.

$B3(\beta)$ requires that the legislator initiating the ballot knows the decree every legislators in the quorum voted for. To do this we will have to exchange messages with the legislators in the quorum.
Let's now address what messages exactly must be sent to conduct a ballot.

- 1: Legislator l chooses to initiate a new ballot, he do so by choosing a new ballot number b among those still available to him and send a NextBallot(b) message to some set of legislators (in practice we will send it to all the legislators)
- 2: Upon receiving a NextBallot(b), legislator m sends a LastVote(b, v) to l where v is the vote with the largest ballot number less than b that m has cast, it is the latest vote from m before ballot b. A vote is composed of a ballot number, a decree and a voter. If m never voted before b we will consider both the ballot number and the decree of the vote to be -1.
  Here, m has to take notes in his ledger to remember the votes he had cast. One thing that we must be careful of is that the set of ballot can change, since legislator l is going to use v as the latest vote from m until b. This means that uppon sending a LastVote(b, v), legislator m MUST NOT cast any vote for ballots between v's ballot and b. Otherwise the value of v used by l will be outdated and we will break the consistency.
- 3: After receiving a LastVote(b, v) message from a majority set Q of legislators, legislator creates a new ballot with number b, quorum Q. The decree d is chosen to satisfie $B3(\beta)$, if no legislator in the quorum casted a vote before this (i.e. there are no successful ballots before b), we will make d being equal to b. It is simply an implementation choice, it could be anything but it is the thing we want to synchronize, so here, we will actually synchronize the number of the first sucessful ballot.
  l then record the new ballot in the notes of his ledger and send a BeginBallot(b, d) to every legislator in Q.
- 4: Upon receiving a BeginBallot(b, v), legislator m decides wether or not to cast his vote in ballot number b. He may refuse to do so if this violate a promise implied by a previous LastVote message. If he decides to vote, he sends a Voted(b, m) message to l and record his vote in his ledger.
  Note that legislator m has the option not to vote if he wants to. Every steps in this protocol are actually facultative, even if not responding to a message will surely prevent progress, it cannot break consistency as it cannot make $B1(\beta) - B3(\beta)$ false. This way, even if a legislator leaves the parliament in the middle of a ballot, it cannot break consistency.
- 5: if legislator l has received a Voted(b, m) message from every legislator m in Q, he writes d in his ledger and sends a Success(d) message to every legislators.
- 6: Upon receiving a Success(d) message, a legislator enters decree d in his ledger.

This is a lot to digest, take a deep breath, we will not dive into the implmentation of this right now, they are some restrictions we can apply in order to ease the implementation of this and I also think it is time to address some design again.
The multiple restrictions will result in the basic protocol so that's where we are going to talk about this.

Concerning design, we can start to think about what the messages we will send will look like.
I decided to use a format really similar to HTTP, first because it allows me to reuse code from another project, but also because it is really easily scalable.
A message will be composed of a method and a set of headers. To put it simply the method will be the type of the message and headers will the parameters. A complete message will look like this:

```
BeginBallot
receiver: m
sender: l
ballot: 4
decree: 2

```

So when I said headers were a set, you can obviously see that I lied, it is in fact a map, it will allow us to easily recover key information in the message. I will not dive into the parsing and the serialization of such messages because we do not care, just know that the end of the message is marked by an empty line.
Also note the receiver header, I put it in every message automatically upon sending it and its value is the name of the legislator who will receive the message, this is solely for debuging and logging purposes, to see where the message is actually going.

5.3 The basic protocol

The preliminary protocol require every legislator to record the number of every ballot he initiated, every vote he has cast and every LastVote message he has sent. That's a lot of information to keep track of without even considering how hard it might be to implement.
To answer this problem, the basic protocol is a restriction of the preliminary protocol.
This means that every action in the basic protocol is allowed by the preliminary protocol (the reverse is wrong), since the preliminary protocol maintains the consistency condition, so does the basic one.

For the basic protocol, every legislator l has to keep track in his ledger of:

- lastTried[l]: The number of the last ballot that l tried to initiate (we will take the liberty to put -1 if no such ballot exists as every ballot has a positive number in our implementation)
- prevVote[l]: The vote cast by p in the highest-numbered ballot in which he voted
- nextBal[l]: The largest value of b for which l has sent a LastVote(b, v)

The preliminary protocol allows legislator l to conduct any number of ballots concurrently, the basic protocol only allow one, he will ignore every message concerning a ballot different from the last he initiated.
In the premilimary protocol a LastVote(b, v) sent by m was the promise to not vote for any ballot between v's ballot and b. In the basic protocol, it represents the stronger promise to not vote for any ballot numbered lower than b. This might prevent m from casting a vote in step 4 that he would have been allowed to cast in the preliminary protocol but as we saw, the preliminary protocol still allows m to not cast a vote even if he is allowed to so the basic protocol is does require m to perform an action forbidden by the preliminary protocol.

It would be pointless to rewrite the previous protocol and simply change some words so I will dive into the code right now, I recommand to take a look at the preliminary algorithm so you can get a better grasp of its progression.
I purposely removed some details in the code (logs, definitions of certain abstractions ...) to make the code lighter, certain abstractions have rather confusing implementations because I'm a terrible developer, but also because they contain optimizations that will be addressed in the complete protocol, yet I think the name of the fonctions will be good enough for you to understand their goal.

5.3.1 Step 1

```cpp
void Legislator::initiate_ballot()
{
    //this attribute keeps tracks of wether or not the ballot has begun
    has_started = false;
    //This attribute is a map of the previous votes of the quorum,
    //it is updated after receiving a LastVote
    quorum_previous_votes.clear();
    int new_ballot_number = get_next_ballot_id();
    //we set lastTried[l] to the newly initiated ballot
    ledger.set_last_tried(new_ballot_number);
    send_next_ballot(new_ballot_number);
}
void Legislator::send_next_ballot(int ballot)
{
    std::string ballot_string = std::to_string(ballot);
    Message message;
    message.set_method("NextBallot");
    message.add_header("ballot", ballot_string);
    message.add_header("sender", self->config_.name);
    for (auto legislator : legislators)
        message.send(legislator.second);
}
```

5.3.2 Step 2

```cpp
//Step 2
void Legislator::receive_next_ballot(Message message)
{
    std::string ballot_str = *message.get_header("ballot");
    int ballot = std::stoi(ballot_str);
    receive_next_ballot(ballot, *message.get_header("sender"));
}

void Legislator::receive_next_ballot(int ballot, std::string sender)
{
    int next_ballot = ledger.next_bal(); //get nextBal[m]
    if (ballot <= next_ballot) //ignore the message if the ballot is outdated
        return;
    ledger.set_next_bal(ballot); //Set nextBal[m] to b
    Vote previous_vote = ledger.prev_vote(); //Get the last vote before b
    send_last_vote(ballot, previous_vote, sender);
}

void Legislator::send_last_vote(int ballot, Vote previous_vote,
        std::string sender)
{
    std::string ballot_string = std::to_string(ballot);
    std::string vote_ballot_id_string
        = std::to_string(previous_vote.ballot_id);
    std::string decree_string = std::to_string(previous_vote.decree.decree);
    Message message;
    message.set_method("LastVote");
    message.add_header("ballot", ballot_string);
    message.add_header("vote_ballot_id", vote_ballot_id_string);
    message.add_header("decree", decree_string);
    message.add_header("sender", self->config_.name);
    message.send(legislators[sender]);

}
```

So here, legislator m receives a NextBallot, if he already responded to a NextBallot from a later ballot (whose ballot number is higher), m discards the message because the ballot is considered to be outdated, as a recall responding to a NextBallot(b) is a promise to never vote for a ballot earlier than b, it only makes sense to discard it now before even be able to vote.
If b is indeed a newer ballot, then m retrieves its last vote and sends it to legislator l.

5.3.3 Step 3

```cpp
void Legislator::receive_last_vote(Message message)
{
    std::string ballot_str = *message.get_header("ballot");
    std::string vote_ballot_id_str = *message.get_header("vote_ballot_id");
    std::string sender =  *message.get_header("sender");
    int ballot = std::stoi(ballot_str);
    int vote_ballot_id = std::stoi(vote_ballot_id_str);
    int vote_decree = std::stoi(*message.get_header("decree"));

    if (ballot != ledger.last_tried() || has_started)
        return;

    Decree decree;
    decree.decree = vote_decree;
    Vote vote;
    vote.decree = decree;
    vote.legislator = sender;
    vote.ballot_id = vote_ballot_id;

    quorum_previous_votes.insert(std::pair<std::string, Vote>
            (sender, vote));
    unsigned int quorum_size = quorum_previous_votes.size();
    unsigned int nb_legislators = legislators.size();
    if (quorum_size > nb_legislators / 2)
        receive_enough_last_vote();
}

void Legislator::receive_enough_last_vote()
{
    has_started = true;
    int ballot = ledger.last_tried();
    std::string ballot_str = std::to_string(ballot);

    Vote max_vote;
    max_vote.ballot_id = -1;
    for (auto legislator_vote_pair : quorum_previous_votes)
    {
        Vote current_vote = legislator_vote_pair.second;
        if (current_vote.ballot_id > max_vote.ballot_id)
            max_vote = current_vote;
    }
    Decree decree;
    if (max_vote.ballot_id != -1)
        decree = max_vote.decree;
    else
        decree.decree = ballot;
    send_begin_ballot(ballot, decree);
}
```

There is actually a lot of things going on here, first I will not show the definition of **send_begin_ballot(int, Decree);** as it is really straightforward, it is basically the same thing than NextBallot but we send it only to the quorum and that we also send a decree.

The first thing we check when we receive a LastVote is wether or not it is for the ballot we are currently conduction (this ballot is marked by lastTried[l]).

We also need to check that we did not already start the ballot, we do not want to send a BeginBallot every time we receive a LastVote when you actually had a large enough quorum to send a BeginBallot beforehand, even though it doesn't impact consistency, it still add too much information in the logs.
After checking the ballot is valid, we register the vote in a map of the legislator names and their respective last vote. We then check the size of the map, if the number of elements is greater than half of the total number of legislators, it means that a majority of legislators responded with a LastVote, we can now enter the actual step 3: find the decree and send a BeginBallot.
We first mark the ballot has started, any incoming LastVote for this ballot will be discarded as the quorum just has beem established.
We then simply check which vote was the latest among the last votes of the quorum and choose the corresponding decree, if no legislator previously voted we arbitrarily choose the ballot number as the new decree.

5.3.4 Step 4

```
void Legislator::receive_begin_ballot(Message message)
{
    std::string ballot_string = *message.get_header("ballot");
    std::string decree_string = *message.get_header("decree");
    std::string sender = *message.get_header("sender");

    int ballot = std::stoi(ballot_string);
    int decree = std::stoi(decree_string);
    receive_begin_ballot(ballot, decree, sender);
}

void Legislator::receive_begin_ballot(int ballot, int decree_id,
        std::string sender)
{
    int next_ballot = ledger.next_bal();
    if (ballot != next_ballot)
        return;
    Vote vote;
    vote.ballot_id = ballot;
    Decree decree;
    decree.decree = decree_id;
    vote.decree = decree;
    ledger.set_prev_vote(vote);

    send_voted(ballot, decree, sender);
}

void Legislator::send_voted(int ballot, Decree decree, std::string receiver)
{
    Message message;
    message.set_method("Voted");
    message.add_header("ballot", std::to_string(ballot));
    message.add_header("sender", self->config_.name);
    message.add_header("decree", std::to_string(decree.decree));
    message.send(legislators[receiver]);
}
```

When receiving a BeginBallot(b, d), we first check if we should cast our vote in the ballot, indeed casting a vote might break a promise from an earlier LastVote, so we need to make sure that b is greater or equal to nextBal[m]. You might notice that we also ignore the message if nextBal[m] is greater than b, the reason is simple: this case is not supposed to ever happen, indeed m can receive a BeginBallot(b,

v) only if m is part of the quorum of b, which mean it sent a LastVote(b) previously which mean that nextBallot[m] was previously set to b (but can be bigger if an other ballot was initiated right after). Even though this case is supposed to never happen, we still handle the possibility by ignoring the message.

Once we checked that we could indeed cast the vote, we cast it and send Voted.

5.3.5 Step 5

```
void Legislator::receive_voted(Message message)
{
    std::string ballot_str = *message.get_header("ballot");
    std::string sender = *message.get_header("sender");
    std::string decree_str = *message.get_header("decree");
    Decree decree;
    decree.decree = std::stoi(decree_str);

    receive_voted(std::stoi(ballot_str), decree, sender);
}

void Legislator::receive_voted(int ballot, Decree decree, std::string voter)
{
    if (ballot != ledger.last_tried())
        return;
    quorum_previous_votes.erase(voter);
    if (quorum_previous_votes.size() == 0)
        receive_enough_voted(ballot, decree);
}

void Legislator::receive_enough_voted(int ballot, Decree decree)
{
    ledger.set_decree(decree);
    send_success(decree);
}

void Legislator::send_success(Decree decree)
{
    Message message;
    message.set_method("Success");
    message.add_header("decree", std::to_string(decree.decree));

    for (auto legislator : legislators)
        message.send(legislator.second);
}
```

When we receive a Voted message for the good ballot, we erase the voter from the quorum, once the quorum is empty, it means that all the quorum has voted, and we can send a Success.

5.3.6 Step 6

```cpp
void Legislator::receive_success(Message message)
{
    std::string decree_str = *message.get_header("decree");
    Decree decree;
    decree.decree = std::stoi(decree_str);
    receive_success(decree);
}

void Legislator::receive_success(Decree decree)
{
    ledger.set_decree(decree);
}
```

Nothing surprising here.

The basic protocol derive directly from the preliminary protocol and maintains the consistency condition, however we have no guarantee of the progress condition as no legislator is forced to ever initiate a ballot.

The complete protocol, derived from the basic protocol, addresses this problem.

5.4 The complete protocol

The progress condition requires that a legislator eventually performs step 1 but we still need to think of the frequency of the ballots. Indeed, initiating too many ballots can prevent progress. If legislator l initiates a ballot number b larger than any other ballot, the LastVote(b, v) from legislator m in step 2 will prevent m from voting in any previously initiated ballot. If new ballots continually initiated with increasing ballot numbers before the previous ballots have a chance to succeed, then no progress can be made.

The main idea introduced by the complete protocol is that only one legislator can initiate ballot, and that only he can decide of the appropriate time to do so.

This legislator is called the president. The election process is irrelevant but it must occur frequently to make sure the president didn't leave the parliament, we obviously need to change the president in such cases.

So as I mentionned earlier the complete protocol is in fact irrelevant in our case, the reason being that for visualisation purposes, I decided to let the ballot initiation process between the hands of the user, to initiate a ballot, you must send a SIGTSTP to the process of the server.

The problem of president and ballot frequency are now completely in the hands of the user who is totally able to simulate the complete protocol by hand.

However, the complete protocol still addresses an interesting optimization regarding presidency change. (in our case, when the user will send ballots from multiple legislators).

This is linked to the way we choose the next ballot number, the basic protocol specifies that legislator l must initiate a ballot with a number higher than lastTried[l], however, by doing so we have no guarantee that this will be higher than an already passed successful ballot that was initiated by someone else.

To handle the latter possibility, legislator l had to learn about any ballot number higher than lastTried[l] that was conducted before he became president.

At the start of the basic protocol I promised an optimization and this what this is all about. Instead of simply taking a ballot number higher than lastTried[l], we actually look if an higher numbered ballot as been initiated by someone else, this ballot number will be stored in nextBal[l].

```cpp
int Legislator::get_next_ballot_id()
{
    int previous_ballot_id = ledger.last_tried();
    if (previous_ballot_id == -1)
        previous_ballot_id = config_.ballot_partition_id - legislators.size();
    int last_voted_ballot = ledger.next_bal();
    while (last_voted_ballot > (int)legislators.size() + previous_ballot_id)
        previous_ballot_id += legislators.size();
    return previous_ballot_id + legislators.size();
}
```

This is only a simple optimization not presented in the original algorithm, the reason it is not presented is because it actually is useless because one problem remains et that response to this problem also handle the above case, but yet, this optimization allow to save a few message sending.

So the problem that remains is that legislator m was conduction ballots while legislator l was outside of the parliament, then legislator l would not have any way to know these ballots in his personnal ledger. If its first action after entering the parliament (before receiving any message) was to initiate decree, he will have a lot of troubles to find a suitable ballot number.

To solve this, the complete protocol proposes to send a message in response to a NextBallot(b) or BeginBallot(b, d) instead of ignoring it when b is inferior or equal to nextBal[m].
This message contains the value of nextBal[m], legislator l, can then update its value of lastTried[l] before re-initiating a ballot.

```cpp
//Is called in receive_next_ballot and receive_begin_ballot
//instead of ignoring the message
void Legislator::send_higher_ballot(int ballot, std::string receiver)
{
    Message message;
    message.set_method("HigherBallot");
    message.add_header("ballot", std::to_string(ballot));
    message.send(legislators[receiver]);
}

void Legislator::receive_higher_ballot(Message message)
{
    std::string ballot_str = *message.get_header("ballot");
    receive_higher_ballot(std::stoi(ballot_str));
}

void Legislator::receive_higher_ballot(int ballot)
{
    int last_tried = ledger.last_tried();
    if (last_tried > ballot)
        return;

    while (last_tried + (int)legislators.size() < ballot)
        last_tried += legislators.size();

    ledger.set_last_tried(last_tried);
    initiate_ballot();
}
```

6 Conclusion and opening on the multi-decree parliament

As I said the implementation is incomplete, I presented here the complete single-decree synod protocol. However the original paper presents the multi-decree parliament protocol which directly derives from the single-decree. Its goal is of course to synchronize not one but multiple decrees between legislators.
This have obviously much more applications than the single-decree protocol but I did not implement it yet, I do not want to risk myself continuing this article on a subject I did not experience myself, but maybe you will be able to understand it by yourself by reading the original paper.
I hope this article gave you the keys you needed to understand the Paxos algorithm with a more pragmatic and intuitive approach.
I will probably implement the multi-decree parliament some day on the same repository, but for now I will have to let you search for it, but do not be scared, it is in fact fairly simple as, I quote,

"The Paxons never bothered writing a precise description of the parliamentary protocol because it was so easily derived from the Synod protocol."